Misc 1.1.5
Reference Manual

Department RIM — Internal Report

Jean-Jacques Girardot
girardot@emse.fr
March 2006

École Nationale Supérieure des Mines de Saint-Etienne
158 Cours Fauriel
42023 Saint-Étienne Cédex

Working draft. Release 0.27
Copyright (c) J.J.Girardot, 2003, 2004, 2005, 2006.
Printed 30 mars 2009
Misc reference manual. This document contains 68 pages.

Documentation is like sex; when it's good, it's very, very good,

and when it's bad, it's better than nothing.

– Dick Brandon

Chapter 1

Introduction

1.1 What is MISC ?

1.1.1 A few words

MISC is a very small implementation of the SCHEME language (hence the name : MISC stands for «Micro scheme») written in Java. Its primary goal was to explain to students what is SCHEME and how to implement it. Although MISC provides only a subset of the SCHEME language, it can be used as an extension language in applications written in Java.

As of version 1.1.5, MISC doesn't provide vectors, neither numeric data types other than 32 bits integers. Also, some operations on ports are not implemented. Conversely, MISC provides a few extensions, that are described in § 2.1.6, starting at page 36. This manual [5] describes the operations of the system. An other manual [4] describes the internals of the system.

1.1.2 About this manual

This is the first released manual of MISC. It is far from being complete. Readers are invited to send their comments to the author, at `girardot@emse.fr`.

MISC was developed on a PC laptop, operating under Linux. This document itself was written using LyX [8], a document editor which provides a quasi-WYSIWYG mode for LaTeX [7].

1.2 A short introduction to MISC

1.2.1 Getting the system

MISC is available from its Home page, at :

`http://kiwi.emse.fr/Misc/`

MISC distribution is a *gzipped tar* file (the name typically being `MiscV1.1.4.tar.gz`), which includes all the java source code, some examples, a test suite, and some documentation.

1.2.2 Installing the system

The installation is quite crude. After unpacking the distribution, with

```
tar xzf MiscV1.1.4.tar.gz
```

go to the directory «`MiscV1.1.4`» and execute

```
make
```

This should be enough to compile

1.2.3 Running the system

The command `java Misc` can be used to run the interpreter. The interpreter displays the prompt «`? `», then expects a SCHEME expression. The result of the evaluation of the expression is displayed, preceded by the string «`=>  `». Here is an example of a session with MISC.

```
[SRC]$ java Misc
Misc V1.1.3 Started
? (define (fib n)
?   (if (< n 2)
?       n
?       (+ (fib (- n 1)) (fib (- n 2)))))
=>  fib
? (fib 10)
=>  55
? (end)
; Cells use : 1749/8192, 0 gc.
Misc ended
[SRC]$
```

1.2.4 About MISC

MISC consists of a set of Java classes that implement the basic structures and the primitive functions of a SCHEME system. SCHEME expressions (*lists*) are first transformed into an internal representation. The interpreter is properly *tail recursive*. It uses for the management of its data structures a *mark and sweep garbage collector*. It fully implements *continuations*, *environments*, and provides *light-weight processes*.

Chapter 2

The language

2.1 A short description of MISC

2.1.1 The standard part of MISC

Generally, MISC tends[1] to conform to the SCHEME language as described in [6]. It uses the Lisp parenthesed syntax, and provides most of the SCHEME data-types. Potential users are encouraged to consult the report [6]. People looking for a good computer science book dealing with SCHEME can consult [1] or its french translation [2].

2.1.1.1 Syntax

MISC expressions use a fully parenthesized prefix notation for programs and data. A Misc program is a sequence of characters that are grouped to constitute tokens, spaces and comments.

Spaces The characters *space*, *tabulation*, *line-feed*, *carriage-return* (and some other control characters) constitute the *space characters*. A sequence of spaces is equivalent to a unique space. Spaces are used to separate syntactic units (symbols, constants). Spaces are not necessary around special characters.

Alphanumeric characters The upper-case and lower-case letters, the digits, and the following characters constitute the alphanumeric characters :

```
! $ % & * + - . / : < = > ? @ ^ _ ~
```

A *symbol* is a non empty sequence of alphanumeric characters, with the following exceptions :

- a non empty sequence of digits, optionally preceded by a − sign, constitute a number ;
- the identifier consisting of the unique character `.` is the *dot*, and is used for the representation of dotted pairs.

[1] But, alas, does not succed...

Special Characters The following characters are considered as «special», and cannot be used inside a symbol.

- *quote* : the quote character `'` indicates that the following form is a literal data. The form can be a constant (in which case the quote character is not necessary), a symbol or a list. The notation **`'item`** is fully equivalent to **`(quote item)`**
- *double quote* : the double quote character `"` is used to delimit strings (see 2.1.4.5).
- *parentheses*, `(` and `)`, are used to denote *lists* (see 2.1.4.9) and *pairs* (see 2.1.4.8).
- *backquote* : the backquote character `` ` `` is used to indicate almost-constant data. The notation **`` `item ``** is fully equivalent to **`(backquote item)`**
- *comma* : the comma character `,`, sometimes followed by the at-sign, `@`, is used in conjunction with backquote. The at-sign is not a «special» character in itself. The notations **`,item`** and **`,@item`** are fully equivalent to **`(unquote item)`** and **`(unquote-splicing item)`** respectively.
- *sharp sign* : the sharp sign character, `#`, is used for a variety of purposes depending on the character following it. In particular, it is used to denote literal booleans and characters.
- *semicolon* : this character, `;`, introduces a comment (see 2.1.1.1).
- *still bar, braces* and *brackets*, `|{}[]` are reserved for future use.

Comments Comments start with a semi-colon, and end at the end of the line, as usual in Lisp-like languages. Note that the end-of-line character itself is not a part of the comment. Comments can appear anywhere in a source file where a space is allowed.

```
; This is a comment
```

Forms A form is the external representation of a SCHEME expression. A form can be :

- a *constant*, such as a *boolean value* (see 2.1.4.1), a *number* (see 2.1.4.2), a *character* (see 2.1.4.4) or a *string* (see 2.1.4.5).
- a *symbol* (see 2.1.4.6) ; symbols are used to denote variables.
- a *list* (see 2.1.4.9) ; lists are used to denote *special forms* and *functions applications*.

Special forms A special form is a list whose first element is an identifier which, in this context, represents a *keyword* of the language. The following table describes the keywords of MISC.

Name	Description	See...
=>	used with *cond*	§ 2.1.2, p. 8
and	conditional expression	§ 2.1.2, p. 8
begin	sequencing	§ 2.1.2, p. 8
case	conditional expression	§ 2.1.2, p. 9
cond	conditional expression	§ 2.1.2, p. 8
define	declaration of variable	§ 2.1.6.1, p. 38
delay	delayed expression	§ 2.1.4.15, p. 31
do	iteration	*not implemented*
else	used with *cond* and *case*	§ 2.1.2, p. 8 and § 2.1.2, p. 9
if	conditional expression	§ 2.1.2, p. 8
lambda	function creation	§ 2.1.4.14, p. 28
let	creation of local environment	§ 2.1.6.1, p. 39
let*	creation of local environment	§ 2.1.6.1, p. 40
letrec	creation of local environment	§ 2.1.6.1, p. 39
or	conditional expression	§ 2.1.2, p. 8
quasiquote	quasi quotation	*not implemented*
quote	quotation	§ 2.1.3, p. 10
set!	assignment of variable	§ 2.1.6.1, p. 38
unquote	quasi quotation	*not implemented*
unquote-splicing	quasi quotation	*not implemented*

Special forms correspond essentially to the control structures of traditional programming languages (sequence, alternative, declarations, assignment...)

Function application A function application is a list whose first element does not denote a keyword of the language. The syntax of such a form is :

$$(exp_0\, exp_1\, exp_2 \ldots exp_n)$$

where the exp_i are SCHEME forms, i.e. constants, symbols or lists.

The SCHEME interpreter evaluates these expressions in an unspecified order, giving a set of values val_0, val_1,... val_n. The first of these values must be a procedure of n parameters. This procedure is then applied to the arguments val_1,... val_n, giving the final result of the application.

2.1.2 Special forms

(if *test consequent alternate*)
(if *test consequent*) The forms *test*, *consequent* and *alternate* may be arbitrary expressions. The *test* is evaluated first. If it yields a *true* value, then *consequent* is evaluated, and its value returned ; otherwise, *alternate* is evaluated and its value returned. If *test* yields a *false* value and no *alternate* is specified, the result of the expression if *false*.

```
? (if (> 2 3) 'yes 'no)
=>  no
? (if #f 1)
=>  #f
```

(and *test*$_1$ ...) The *test* expressions are evaluated from left to right, and the value of the first expression that evaluates to a *false* value is returned. Any remaining expressions are not evaluated. If all expressions evaluate to *true* values, the value of the last expression is returned. If there are no expressions, then `#t` is returned.

```
? (and (> 5 2) (< 5 10))
=>  #t
? (and (= 2 2) (< 2 1))
=>  #f
? (and 1 2 'c '(d e f))
=>  (d e f)
? (and)
=>  #t
```

(or *test*$_1$ ...) The *test* expressions are evaluated from left to right, and the value of the first expression that evaluates to a *true* value is returned. Any remaining expressions are not evaluated. If all expressions evaluate to *false* values, this value is returned. If there are no expressions, then `#f` is returned.

```
? (or (> 2 4) (< 2 6))
=>  #t
? (or 2 (quotient 3 0))
=>  2
? (or)
=>  #f
```

(begin *exp*$_1$ *exp*$_2$ ... *exp*$_n$) This is a sequencing form, which evaluates the forms exp_1, exp_2 up to exp_n in this order, and returns the result of the last form evaluated.

```
? (begin 1 2 3)
=>  3
? (begin (write "hello") (newline) (+ 3 4))
"hello"
=>  7
```

(cond *clause*$_1$ *clause*$_2$ ...) This is a conditional expression, where each ***clause***$_i$ may have one of the following syntaxes :

1. **(*test expression*$_1$ ...)**
2. **(*test* => *expression*)**
3. **(else *expression*$_1$ ...)**

A *cond* expression is evaluated by executing the *tests* expressions in the successive *clauses* until one of them evaluates to a *true* value. The remaining expressions in the *clause* are then evaluated, and the result of the last one is returned as the result of the *cond* form. If there is no *expression* in the *clause*, the result of the *test* is returned.

```
? (cond (#f 1) (#t 2) (#t 3))
=>  2
? (cond (#f 1 2 3) (4 5 6 7))
=>  7
? (cond ((member '(a) '((b) (a) (c)))))
=>  ((a) (c))
```

If the selected *clause* uses the **=>** form, then the corresponding expression is evaluated ; its value must be a procedure which accepts one argument ; this procedure is then applied on the value returned by the *test*, and the result of this application is returned as the result of the form.

```
? (cond ((member '(a) '((b) (a) (c))) =>  car))
=>  (a)
```

The *else* form of the clause can only be used as the last clause of a *cond*, and is equivalent to a clause whose *test* is always *true*. The $expressions_i$ are then executed in sequence, and the result of the last one is returned as the result of the *cond* form.

```
? (define x -3)
=>  x
? (cond ((> x 0) 'positive) ((= x 0) null) (else 'negative))
=>  negative
```

(case *key* *clause*$_1$ *clause*$_2$ ...) This is a conditional expression, where the ***key*** may be any expression, and each $\boldsymbol{clause}_i$ may have one of the following syntaxes :

1. **((*datum*$_a$ *datum*$_b$ ...) *expression*$_1$ ...)**
2. **(else *expression*$_1$ ...)**

The $\boldsymbol{datum}_i$ are external representations of SCHEME objects. The value of the ***key*** is searched in the $\boldsymbol{datum}_i$ of each ***clause***. If the ***key*** is equivalent (in the sense of the ***eqv?*** procedure) to one the $\boldsymbol{datum}_i$, the corresponding ***clause*** is selected, all the $\boldsymbol{expression}_k$ are evaluated in sequence, and the value of the last one is returned as the result of the ***case*** form.

The second syntax of the ***clauses*** can only be used as the last of the clauses. It is selected if none of the preceding clauses have been selected.

```
? (case 'a ((a b c d) 1) ((e f g a) 2) ((h i j k l) 3))
=>  1
? (case 'k ((a b c d) 1) ((e f g a) 2) ((h i j k l f a) 3))
=>  3
? (case 'p ((a b c d) 1) ((e f g) 2) ((h i j k l) 3))
=>  #f
? (case 'p ((a b c d) 1) ((e f g) 2) ((h i j k l) 3) (else 4))
=>  4
? (case 'q (() 2) (else 3))
=>  3
? (case (car '(c d))
?       ((a e i o u) 'vowel)
?       ((w y) semi-vowel)
?       (else 'consonant))
=>  consonant
```

2.1.3 Data types

Misc implements the following data-types :

- *booleans* represent truth values, *true* or *false*. In conditional expressions, any SCHEME value (including the empty list) different from the boolean *false* is considered *true*.
- *numbers* represent the numeric values.
- *characters* represent printed characters, such as letters or digits.
- *strings* are sequences of characters.
- *symbols* represent both identifiers of the language and symbolic data.
- *procedures* represent algorithms, either predefined in the language, or dynamically defined by the user.
- the *empty list* is a specific distinguished value used in the representation of lists.
- *pairs* are record structures with two fields called *car* and *cdr*. Pairs are used in the representation of lists.
- *vectors* are structures that can handle an arbitrary number of objects. Items in a N elements vector are referenced by an integer comprised between 0 and $N-1$.
- *ports* represent input and output devices.
- *environments* are sets of name/value couples representing the variables of the MISC system.
- *continuations* represent the *future* of a computation.
- *threads*, *locks*, *cells* and *barriers* are data-types used in the management of light weight processes in MISC.

All these objects are *first class citizen*, which means they can be dynamically created or *reified* by system procedures, tested for specific properties, passed as parameters to procedures, returned from procedures, and used as items in MISC data structures. Since some data types (symbols, lists) play a syntactic role in the language, they need to be *quoted* to be considered as data rather than program expressions. This is the purpose of the quote special form.

'object
(quote object) This special form returns its operand unevaluated. It is used to include literal constants, such as symbols and lists, in Scheme code. The two notations are equivalent, the first one being transformed into the second one by the Scheme reader.

```
? 'hello
=>  hello
? (quote hello)
=>  hello
? "hello
=>  'hello
```

```
? (car "hello)
=>  quote
? (equal? "hello '(quote hello))
=>  #t
```

2.1.4 Operations on data types

2.1.4.1 Booleans

The values *true* and *false* are represented by the notations `#t` and `#f` respectively. The following operations are related to the boolean data type :

(boolean? *object*) This predicate returns *true* if its parameter is a boolean value, *false* otherwise.

```
? (boolean? #t)
=>  #t
? (boolean? "Hello")
=>  #f
```

(not *object*) This operation returns *true* if its parameter is the boolean value *false*, *false* otherwise.

```
? (not #f)
=>  #t
? (not "Hello")
=>  #f
```

2.1.4.2 Numbers

In this implementation, integers represent the only numeric data-type implemented.

(number? *object*) This predicate returns *true* if its parameter is a number, *false* otherwise.

2.1.4.3 Integers

Integers are represented by a non empty sequence of decimal digits, optionally preceded by a minus sign. Integers are constants and do not need to be quoted. Integers are represented by Java 32 bits integers, and have values comprised between -2^{31} and $2^{31}-1$. The following operations are related to the integer data type :

(integer? *object*) This predicate returns *true* if its parameter is an integer value, *false* otherwise.

(+ *number* ...) This operation returns the sum of its parameters.

```
? (+ 2 5 9 12)
=>  28
? (+)
=>  0
? (+ 2)
=>  2
```

(* *number* . . .) This operation returns the product of its parameters.

```
? (*)
=>  1
? (* 2 3 5 8)
=>  240
```

(- *number* . . .) With one argument, this operation returns the opposite of its parameter. With two or more arguments, it returns the difference between the first parameter and the sum of the others.

```
? (- 7)
=>  -7
? (- 7 1)
=>  6
? (- 7 1 2 3)
=>  1
```

(> *number number*)
(>= *number number*)
(<= *number number*)
(< *number number*)
(= *number number*)
(!= *number number*) These operations compare their arguments. The result is a boolean value, *true* or *false*.

```
? (> 5 2)
=>  #t
? (<= 8 8)
=>  #t
? (= 2 7)
=>  #f
```

(positive? *number*)
(zero? *number*)
(negative? *number*) These predicates compare their argument (which must be numeric) to zero, and return a boolean value.

```
? (positive? 78)
=>  #t
? (negative? 67)
=>  #f
? (zero? 0)
=>  #t
```

(odd? *number*)
(even? *number*) These predicates test the parity of their argument (which must be numeric), and return a boolean value.

```
? (even? 2743)
=>  #f
? (odd? 2743)
=>  #t
```

(abs *number*) This operation returns the absolute value of its parameter.

```
? (abs 245)
=>  245
? (abs -245)
=>  245
```

(max *number* . . .) This operation returns the maximum of its parameters.

```
? (max 2 3 9 5)
=>  9
? (max)
=>  -2147483648
```

(min *number* . . .) This operation returns the minimum of its parameters.

```
? (min 2 3 9 5)
=>  2
? (min)
=>  2147483647
```

(quotient *number number*)
(remainder *number number*)
(modulo *number number*) These operations return the *quotient*, the *remainder* and the *modulo* of their operands. Results are integers.

```
? (quotient 13 4)
=>  3
? (modulo 13 4)
=>  1
? (remainder 13 4)
=>  1
```

When the result is not null, its sign is identical to the sign of the *dividend* for the ***remainder*** operation, and to the sign of the *divisor* for the ***modulo*** operation.

```
? (modulo 13 -4)
=>  -3
? (remainder 13 -4)
=>  1
```

Note Misc introduce the possibility to represent integer numbers coded in other bases than 10. More specifically, the prefixes #B, #O, #D, #X (and the corresponding lower case letters #b, #o, #d and #x) allow to write numbers in base 2, 8, 10 or 16. In the latter case, characters «A» to «F» (and «a» to «f») are used for the «digits» greater than 9 :

```
? #b101
;=>  5
? #o101
;=>  65
? #d101
;=>  101
? #x101
;=>  257
? #x2a
;=>  42
```

2.1.4.4 Characters

The notation `#\char` generates a character. A character is a constant and doesn't need to be quoted. When displayed as a result of a computation, characters are printed in a way suitable for a further input by the interpreter.

```
? #\a
=>  #\a
```

(char? *object*) This predicate returns *true* if its parameter is a character, *false* otherwise.

```
? (char? #\a)
=>  #t
```

(integer->char *object*) The parameter is an integer which represents the index in the Unicode table of the desired character. Such a character may not be printable on every output device.

```
? (integer->char 545)
=>  #\?
```

(char->integer *object*) This predicate returns an integer whose value is the index of the character in the Unicode table.

```
? (char->integer #\ç)
=>  231
```

2.1.4.5 Strings

Strings are represented as (possibly empty) sequences of characters enclosed between double quotes (`"`). Double quotes and backslashes can be written inside a string by preceding them with a backslash. Thus, `"\"\\"` is a length 2 character string consisting of a double quote followed by a backslash. Backslash followed by the character **n** denotes a line-feed ; backslash followed by a line feed is suppressed from the string. A string is a constant and doesn't need to be quoted. When displayed as a result of a computation, strings are printed in a way suitable for a further input by the interpreter.

```
? "abcd\
? efgh"
=>  "abcdefgh"
? "\"\\\n"
=>  "\"\\\n"
? (display "\"\\\n")
"\
=>  "\"\\\n"
? (write "\"\\\n")
"\"\\\n"=>  "\"\\\n"
```

(string? *object*) This predicate returns *true* if its parameter is a string, *false* otherwise.

```
? (string? "abcd")
=>  #t
? (string? 2534)
=>  #f
```

(string *object* ...) This function returns a string constituted by the concatenation of its parameters. Numbers are edited according to their decimal representation.

```
? (string #\a #\b #\c #\d)
=>  "abcd"
? (string)
=>  ""
? (string "hello" 32 "world!")
=>  "hello32world!"
```

(string-append *object* ...) This function returns a string constituted by the concatenation of its parameters. Non strings arguments are first converted to string using the **string** operation.

```
? (string-append #\a #\b #\c #\d)
=>  "abcd"
? (string-append "hello" 32 "world!")
=>  "hello32world!"
```

In this implementation, *string-append* is just another name for *string*.

(string-length *object*) This function returns the number of characters of its parameter.

```
? (string-length "hello")
=>  5
```

(make-string *size {value}*) This function returns a string of *size* characters. The optional second parameter is used to fill up the string.

```
? (make-string 10)
=>  "          "
? (make-string 10 #\*)
=>  "**********"
```

(number->string *number*) This function returns a string of characters representing the decimal form of the number.

```
? (number->string (* 132 (+ 12 89)))
=>  "13332"
```

(string-ref *string number*) This operation returns the character of the string (the first argument) designated by the index (the second argument). Index origin 0 is used.

```
? (string-ref "abcde" 3)
=>  #\d
```

(string-set! *string number char*) This operation replaces a character of the first argument (a string), at the position specified by the second argument (a number), by the third argument (a character or a number). The result is the previous value. As in **string-ref**, index origin 0 is used.

```
? (define s "abcd")
=>  s
? (string-set! s 2 #\!)
=>  #\c
? s
=>  "ab!d"
```

(substring *string start end*) This operation returns a substring of the first parameter (a string). The substring begins with index ***start*** (inclusive) and ends with index ***end*** (exclusive).

```
? (substring "abcdefg" 3 6)
=>  "def"
```

(string-match *pattern string*) This operation implements a very simple *pattern matching* algorithm, where the character «⋆» is the only wild-card character, meaning «any sequence, possibly empty, of characters». The pattern is the first argument, the string the second one.

```
? (string-match "a⋆c" "abc")
=>  #t?
(string-match "a⋆c" "abcbcabcbac")
=>  #t?
(string-match "a⋆c" "ac")
=>  #t?
(string-match "a⋆c" "xyz")
=>  #f
```

2.1.4.6 Symbols

A symbol is represented by a sequence of characters that does not include the special delimiters of the SCHEME language. Used inside an expression, symbols represent variables of the language. They need to be quoted to be interpreted as symbolic data.

```
? toto
Error : toto undefined
? 'toto
=>  toto
? (quote toto)
=>  toto
```

(symbol? *object*) This predicate returns *true* if its parameter is a symbol, *false* otherwise.

```
? (symbol? 'abcd)
=>  #t
? (symbol? "abcd")
=>  #f
```

(string->symbol *string*) This function returns the symbol whose print name is the string passed as argument.

```
? (string->symbol "Nabuchodonausore")
=>  Nabuchodonausore
? (eq? 'Nabuchodonausore (string->symbol "Nabuchodonausore"))
=>  #t
```

(symbol->string *symbol*) This function returns a string corresponding to the print name of the symbol. Modifying the string doesn't change the symbol's name.

```
? (define toto 'titi)
=>  toto
? (define str (symbol->string toto))
=>  str
? str
=>  "titi"
? (string-set! str 2 #\!)
=>  #\t
? str
=>  "ti!i"
? toto
=>  titi
```

2.1.4.7 Empty list

The empty list is the only element of the *empty list* data type. The empty list is represented by an open parenthesis followed by a closing parenthesis (). It is different from any other object. The empty list is a constant and doesn't need to be quoted.

```
? (if () 1 2)
=>  1
```

As this example shows, the empty list isn't considered as false.

(null? *object*) This predicate returns *true* if its parameter is the *empty list*, *false* otherwise.

```
? (null? ())
=>  #t
? (null? #f)
=>  #f
? (null? 123)
=>  #f
```

2.1.4.8 Pairs

A pair is a record structure with two fields, called the *car* and the *cdr*. A pair is created by the `cons` function. The *car* and *cdr* fields are accessed by the `car` and `cdr` functions. The *car* and *cdr* fields are modified by the `set-car!` and `set-cdr!` functions.

In a program, a *pair* can be represented by an open parenthesis, followed by a representation of the value of its *car*, a *dot* symbol, a representation of the value of its *cdr*, and a final closing parenthesis. Since pairs are also used to represent expressions, a pair needs to be quoted to be interpreted as a data.

```
? '(2 . x)
=>  (2 . x)
```

When printing a *pair*, the interpreter uses the following convention : if a *dot* is to be printed followed by an open parenthesis, the interpreter suppresses, in its printed output, the *dot*, the opening parenthesis and the matching closing parenthesis :

```
? '((a . b) . (c . d))
=>  ((a . b) c . d)
? '(a . (b . (c . (d . ()))))
=>  (a b c d)
```

(pair? *object*) This predicate returns *true* if its parameter is a pair, *false* otherwise.

```
? (pair? 'toto)
=>  #f
? (pair? '(a b))
=>  #t
? (pair? '())
=>  #f
```

(cons *object object*) This operation returns a new pair, whose *car* is the first parameter, and whose *cdr* is the second parameter.

```
? (cons 3 5)
=>  (3 . 5)
? (cons 3 '(a b c))
=>  (3 a b c)
```

(car *object*) This operation returns the *car* of its parameter, which must be a pair.

```
? (car '(a . b))
=>  a
? (car '(1 2 3 4))
=>  1
```

(cdr *object*) This operation returns the *cdr* of its parameter, which must be a pair.

```
? (cdr '(a . b))
=>  b
? (cdr '(1 2 3 4))
=>  (2 3 4)
```

(set-car! *object value*)
(set-cdr! *object value*) These operations modify the *car* (respectively *cdr*) of a pair (the first parameter), by replacing it by the value passed as the second argument. The operations return the (modified) pair.

```
? (define p (cons 'a 'b))
=>  p
? p
=>  (a . b)
? (set-car! p "Hello")
=>  ("Hello" . b)
? (set-cdr! p "World")
=>  ("Hello" . "World")
? p
=>  ("Hello" . "World")
```

(caar *object*)
(cadr *object*)
(cdar *object*)
(cddr *object*) These operations return combinations at 2 levels of operations ***car*** and ***cdr***.

```
? (cadr '(1 2 3 4))
=>  2
? (car (cdr '(1 2 3 4)))
=>  2
```

(caaar *object*)
(caadr *object*)
(cadar *object*)
(caddr *object*)
(cdaar *object*)
(cdadr *object*)
(cddar *object*)
(cdddr *object*) These 8 operations return combinations at 3 levels of operations *car* and *cdr*.

(caaaar *object*)
(caaadr *object*)
...
(cddddr *object*) These 16 operations return combinations at 4 levels of operations *car* and *cdr*.

2.1.4.9 Lists

Lists are a data-type represented by the union of the *empty list* and a subset of pairs. More precisely, a list is defined as an object which is either the *empty list* or a pair whose *cdr* is a list.

(list? *object*) This predicate returns *true* if its parameter is a *proper list*, i.e. an object which is either the *empty list* or a pair whose *cdr* is a proper list.

```
? (list? '(a b c d))
=>  #t
? (list? '())
=>  #t
? (list? '(a . b))
=>  #f
? (let ((u (list 'a)))
?       (set-cdr! u u)
?       (list? u))
=>  #f
```

(list *object* ...) This operation returns the list of its parameters. This function generates as many pairs as there are arguments. The result can be viewed as a set of ***cons*** operations.

```
? (list 'a 'b 'c 'd)
=>  (a b c d)
? (cons 'a (cons 'b (cons 'c (cons 'd '()))))
=>  (a b c d)
```

(append *object* ...) This operation returns the list consisting of the elements of the first parameter, followed by the elements of the other parameters. All arguments, except the last, should be lists. Arguments are not modified by the operation.

```
? (append '(a b c) '(x y))
=>  (a b c x y)
? (append '() 'a)
=>  a
? (append '(x y) 'a)
=>  (x y . a)
```

(append! *object* ...) This operation modifies its firsts parameters (which must be lists), to return the list obtained by physically appending the other parameters to the first one. All arguments, except the last, should be lists, which are modified in the process.

```
? (define l '(a b c d))
=> l
? l
=> (a b c d)
? (append! l '(x y z))
=> (a b c d x y z)
? l
=> (a b c d x y z)
? (append! l 333)
=> (a b c d x y z . 333)
? (append! l ())
=> (a b c d x y z)
```

(reverse *list*) This operation returns a new created list, that contains the same items as its parameter (which must be a list), in the reverse order. The parameter is not modified by this operation

```
? (define l '(a b c d))
=>  l
? (reverse l)
=>  (d c b a)
? l
=>  (a b c d)
```

(reverse! *list*) This operation modifies its parameter (which must be a list), to rearrange its items in the reverse order. The parameter is physically modified by this operation

```
? (define l '(a b c d))
;=>  l
? (define m (reverse! l))
;=>  m
? m
;=>  (d c b a)
? l
;=>  (a)
? (reverse! m)
;=>  (a b c d)
? m
;=>  (d)
? l
;=>  (a b c d)
```

(length *object*) This operation returns the length of the list passed as argument. The object must be a proper list.

```
? (length '(a b c d))
=>  4
? (length '())
```

```
=>  0
? (length '(a . b))
Error : list expected
```

(copy *list*) This operation returns a new created list which is a copy of the parameter. The operation `(copy x)` is equivalent to `(append x '())` or `(reverse! (reverse x))`. Note that the result may share some elements with the original object.

```
? (define s '(a (b c) (d e f)))
;=>  s
? (define t (copy s))
;=>  t
? (cadr s)
;=>  (b c)
? (eq? (cadr s) (cadr t))
;=>  #t
```

(deep-copy *list*) This operation execute a copy of the whole structure passed as parameter. Contrary to the copy operation, no element of the result is shared with the original.

```
? (define v (deep-copy s))
;=>  v
? v
;=>  (a (b c) (d e f))
? (eq? (cadr s) (cadr v))
;=>  #f
```

(list-tail *list k*) This operation returns the sublist of *list*, obtained by omitting the first *k* elements. The operation doesn't modify its parameter. It is an error if the list has fewer than *k* elements.

```
? (define l '(a b c d e f g h i j))
=>  l
? (list-tail l 0)
=>  (a b c d e f g h i j)
? (list-tail l 3)
=>  (d e f g h i j)
? (list-tail l 12)
Error : pair expected
```

(list-ref *list k*) This operation returns the *k*th element of *list*. It is an error if the list has fewer than *k* elements.

```
? (define l '(a b c d e f g h i j))
=>  l
? (list-ref l 0)
=>  a
? (list-ref l 3)
```

```
=>  d
? (list-ref l 12)
Error : pair expected
```

2.1.4.10 Vectors

A vector is a data-structure that contains a sequence of ordered items, which can be any Misc object.

(vector *elt*1 *elt*2 *elt*3 ... *elt*n) This operation returns a n-elements vector, whose items are the values of the parameters :

```
? (define v (vector 2 (* 3 4) 'toto "hello"))
;=>  v
? (vector? v)
;=>  #t
? (vector-ref v 3)
;=>  "hello"
? (vector-ref v 1)
;=>  12
```

(make-vector *n val*) This operation returns a *n* elements vector. The second parameter is optional, and represents the value that will be used to fill the vector. By default, a value of «0» is used.

```
? (define w (make-vector 100 33))
;=>  w
```

(vector? *exp*) This operation returns *true* if its parameter is a vector, *false* otherwise.

(vector-length *vec*) This operation returns the length of its parameter, which should be a vector.

```
? (vector-length v)
;=> 4
? (vector-length w)
;=>  100
```

(vector-ref *vec n*) This operation returns the value of element *n* of the vector *vec*.

```
? (vector-ref w 27)
;=>  33
? (vector-ref v 3)
;=>  "hello"
```

(vector-set! vec n value) This operation replaces the *nth* element of vector *vec* by its third parameter.

```
? (vector-set! v 0 (cons 'a 'b))
;=> (a . b)
? (cdr (vector-ref v 0))
;=> b
```

(vector-fill! vec value) This operation replace all elements of the vector *vec* by the second parameter. The size of the vector is not modified by the operation.

```
? (vector-fill! v 'hello)
;=>  #<VECTOR:4>
?  (vector-ref v 0)
;=>  hello
? (vector-ref v 3)
;=>  hello
```

2.1.4.11 Ports

Ports are the objects on which SCHEME performs its input and output operations. Ports are devices that accept commands to deliver (input ports) or accept (output ports) characters.

(port? *object*)
(input-port? *object*)
(output-port? *object*) These predicates return *true* if their parameter is a *port*, or more specifically, a *port* than can be used for input or output.

terminal-input-port
terminal-output-port
trace-port These variables designate the default ports on which MISC performs its input, output and trace operations.

sink-input-port
sink-output-port These variables designate *sink* ports. A *sink* output port accepts (and ignores) any character ; a *sink* input port delivers end-of-file objects.

(current-input-port)
(current-output-port) These functions return the ports currently used as default ports for input and output operations.

(open-input-string *string*) This function creates an input port that provides the successive characters of the string passed as parameter.

```
? (define pp (open-input-string "23 (a b c) end"))
=>  pp
? (read pp)
=>  23
? (read pp)
=>  (a b c)
? (read pp)
=>  end
? (read pp)
=>  #<EOF>
```

(open-output-string) This function creates an output port that accepts characters and accumulates them in a buffer.

(get-port-string *port*) The parameter must be a port created with `open-output-string`. The function returns as a string the characters accumulated so far by the successive *write* operations on the port.

```
? (define sp (open-output-string))
=>  sp
? (begin (write "Hello" sp) (newline sp))
=>  #t
? (begin (display "Hello" sp) (newline sp))
=>  #t
? (get-port-string sp)
=>  "\"Hello\"\nHello\n"
? (display (+ 2 3 5 8) sp)
=>  18
? (get-port-string sp)
=>  "\"Hello\"\nHello\n18"
```

(close-input-port *port*) This function closes the specified input port. Although it is an error to read from a closed input port, MISC actually delivers the *end-of-file object* in such a case.

(close-output-port *port*) This function closes the specified output port. It is an error to write to a closed output port.

(open-input-file *string*) This operation creates an input port which delivers the successive characters of the file whose name is passed as a parameter. An error is signalled if the file doesn't exist, or can't be opened for input.

(open-output-file *string*) This operation creates an output port which records the characters written to this port in a file whose name is passed as a parameter. An error is signalled if the file can't be created.

(open-input-sink) This operation creates an input port which delivers *end-of-file*.

(open-output-sink) This operation creates an output port which accepts and discards any character written to this port.

(open-URL *string*) This operation creates an input port which delivers the successive characters of the internet resource whose name is passed as a parameter. An error is signalled if the object doesn't exist, or can't be opened for input.

2.1.4.12 Input operations

(read *{port}*) This function returns a SCHEME object whose external representation has been read on the specified port. In some cases, this function may return a *read-error* or an *end-of-file object*.

(read-token *{port}*) This non-standard function returns an object that represents the next token that can be read on the specified ports. Tokens are simple objects (numbers, atoms, chars, strings), special characters (parentheses, dot, quote, back-quote, comma), or special objects (*read-error* or *end-of-file).*

(read-char *{port}*) This function returns a character read on the specified input port. In some cases, this function may return a *read-error* or an *end-of-file* object.

```
? (read-char)+
=>  #\+
? (list (read-char)(read-char))ab
=>  (#\b #\a)
```

(peak-char *{port}*) This function returns the character available on the specified input port. The character is not withdrawn from the input stream. In some cases, this function may return a *read-error* or an *end-of-file* object.

```
? (peak-char)+
=>  #\+
=>  #<SUBR:Sch.MathsOps:+>
```

In this example, the **+** character is returned as a result of the function, but is left in the input stream, and is therefore read again by the *Read-Eval-Print* loop, this time as the symbol **+**, whose value is the addition primitive operation.

(char-ready? *{port}*) This function returns the boolean *true* if a *read-char* or a *peak-char* executed on the given port can be satisfied because a character is readily available in the input buffer, and false if a *read-char* or a *peak-char* would imply an actual input operation.

```
? (list (char-ready?) (read-char))
=>  (#f #\newline)
? (list (char-ready?) (read-char))+
=>  (#t #\+)
```

In the first case, the *newline* character is the only one available. After the *read-char* is performed, no character is available, and *char-ready?* returns *false*. In the second case, both a *plus* character and a *newline* character are available, and one character, the *newline*, is still available after the *read-char* is performed. *char-ready?* therefore returns *true*. Note that the function returns *true* when the port is in *eof* state.

(read-line *{port}*) This non standard function returns a line (i.e. a sequence of characters terminated by a line-feed or an end of file, but excluding this character) on the specified input port. The result is a string, although in some cases this function may return a *read-error* or an *end-of-file* object.

```
? (read-line)
=>  ""
? (read-line)this will be read
=>  "this will be read"
```

(eof-object? *object*) This function returns *true* if its parameter is the *eof object* (obtained by a call to *read*, *read-token* or *read-char*).

(load *object*) This function evaluates expressions read from the source designated by its parameter, until an end of file is reached. The parameter can be a string (representing the name of an input file), or an input port. The function returns *true.*

```
? (load "essai.scm")
=>  #t
? (load "truc.scm")
Error : can't open file
? (load (open-input-string "(define abc '(a b c))"))
=>  #t
? abc
=>  (a b c)
? (load (open-URL "http://kiwi.emse.fr/Misc/stdlib.scm"))
=>  #t
```

(read-XML *{port}*) This operation reads an XML element on the specified port, or on standard input if no port is specified. The result is a list that reflects the structure of the element that has been read : the car is the open tag, with the name of the tag as first element, and a list of dotted pairs (symbol / values) that reflect the values of the attributes ; the cdr of the result is a sequence that describes the contents. In this sequence, elements are represented by lists, text by strings.

```
? (read-XML)
? <element>hello
? world</element>
;=>  ((element) "hello\nworld")
? (read-XML)
? <abc/>
;=>  ((abc))
? (read-XML)
? <hello who='world'/>
;=>  ((hello (who . "world")))
? (read-XML)
? <an-element attribute1="the value"
?     att2='something else'>
?     the contents <nested/></an-element>
;=>  ((an-element (attribute1 . "the value")
     (att2 . "something else")) "\n    the contents " ((nested)))
```

(read-XML-token *{port}*) This operation reads a single XML token on the specified port, or on standard input if no port is specified. A token can be an opening tag, a space inside a tag, a part of an attribute, etc.

```
? (list (read-XML-token) (read-XML-token))<abc/>
;=>  (#<SPEC:18:0> "<abc")
```

(This example reminds us that operands of a function are evaluated from right to left).

2.1.4.13 Output operations

(write *object {port}*) This function writes its first argument (any object), on the device defined by the optional second argument, by default the terminal output port. Writing is done is a way that is suitable for further input by the read operation.

```
? (write "Hello")
"Hello"
=>  "Hello"
? (write #\a)
#\a
=>  #\a
```

(display *object {port}*) This function writes its first argument (any object), on the device defined by the optional second argument, by default the terminal output port. Writing is done is a way that is suitable for reading by a human being.

```
? (display "Hello")
Hello
=>  "Hello"
? (display #\a)
a
=>  #\a
```

(newline *{port}*) This function prints a new line on the specified port.

(the-non-displayable-object) This function returns an object which, specifically, won't be printed by the output primitive, except when it is the part of a structure. This is a way to avoid printing the result of a function.

```
? (the-non-displayable-object)
? (display (the-non-displayable-object))
? (write (the-non-displayable-object))
? (list 1 (the-non-displayable-object) 2 3)
=>  (1 #<UNDISP> 2 3)
```

2.1.4.14 Procedures

The term of procedure is used to reference *primitive* and *defined operations*, and also *continuations* which are described in an other part of this document (see § 2.1.4.16, page 32).

Procedures are *first-class citizen* in SCHEME.

(lambda *formals body*) This special form creates a new procedure. ***Formals*** is a set of symbols, that denotes the *formal* arguments of the procedure. ***Body*** is a list of forms, that represent the *body* of the procedure. The environment in effect when the *lambda expression* was evaluated is remembered as part of the procedure. When the procedure is later called with actual arguments, a new environment is created, which extends the environment of the procedure with new bindings corresponding to the association of the names appearing in the *formal arguments* and the values provided as the *actual arguments*.

The body of the procedure is then evaluated in this new environment, and the result of the last form of the body is returned as the result of the procedure.

The *formals* can be a list of symbols, a single symbol, or an improper list ending with a symbol :

- **(variable$_1$ variable$_2$ ... variable$_n$)** : the procedure accepts a fixed number of parameters, n, and each of the symbols of the list denotes an actual parameter when the function is called.

```
? ((lambda (x) (+ x 1)) 4)
=>  5
? ((lambda () 4))
=>  4
? ((lambda (x y z) (+ x y z)) 1 2 3)
=>  6
```

- **variable** : the procedure accepts an arbitrary number of parameter, possibly zero, and the single symbol denotes the list of the values of the parameters :

```
? ((lambda x x) 2 3 5)
=>  (2 3 5)
? ((lambda x x))
=>  ()
? ((lambda x x) (+ 2 3) 4 (* 5 6 7))
=>  (5 4 210)
```

- **(variable$_1$ variable$_2$ ... variable$_n$. rest)** : the procedure expects at least n parameters, and additional values are gathered as a list in the variable *rest*.

```
? ((lambda (a b c . z) (list a b c z)) 1 2 3)
=>  (1 2 3 ())
? ((lambda (a b c . z) (list a b c z)) 1 2 3 4)
=>  (1 2 3 (4))
? ((lambda (a b c . z) (list a b c z)) 1 2 3 4 5 6 7 8)
=>  (1 2 3 (4 5 6 7 8))
```

(procedure? *object*) This predicate returns *true* if its parameter is a procedure. The predicate recognizes primitive operations, defined procedures and continuations.

```
? (procedure? car)
=>  #t
? (procedure? (lambda (x) (+ x 1)))
=>  #t
? (call-with-current-continuation procedure?)
=>  #t
? (procedure? if)
=>  #f
```

(apply *procedure {argument . . . } list*) This operation applies its first argument (which must be a procedure) to a list of parameters obtained by consing the optional arguments in front of the last parameter, which must be a list.

```
? (apply + '(2 3 4))
=>  9
? (apply + 2 3 '(4))
=>  9
? (apply cons '(a b))
=>  (a . b)
? (apply cons 'a 'b '())
=>  (a . b)
? (apply apply apply + 2 3 4 '((())))
=>  9
```

(map *procedure list list . . .) This operation takes a first argument, which must be a procedure, and at least one list of elements. It applies the procedure to sets of objects, consisting of the first elements of each list, then the second elements of each list, up to the last elements of each list. The different lists should have the same number of elements. The result is the list of these applications.

```
? (map - '(2 3 5 6 9))
=>  (-2 -3 -5 -6 -9)
? (map + '(2 3 5 6) '(3 5 7 9))
=>  (5 8 12 15)
? (map * '(1 2 3) '(2 0 4) '(3 5 8))
=>  (6 0 96)
? (map (lambda (x) (+ x 1)) '(2 3 5 6))
=>  (3 4 6 7)
```

(for-each *procedure list list . . .) This operation is similar to *map*. It takes a first argument, which must be a procedure, and at least one list of elements. It applies the procedure to sets of objects, consisting of the first elements of each list, then the second elements of each list, up to the last elements of each list. The difference with map is that no result is gathered, and the value **`#f`** is returned. The operation is therefore used for its side-effects.

```
? (for-each + '(2 3 5 6) '(3 5 7 9))
=>  #f
? (for-each write '(a b c d))
a b c d
=>  #f
? (define counter 0)
=>  counter
? (define (add-counter x) (set! counter (+ counter x)))
=>  add-counter
? counter
=>  0
? (for-each add-counter '(2 3 4))
```

```
=>  #f
? counter
=>  9
```

2.1.4.15 Promises

A promise is a computation that has been delayed until its value is needed. The special form *delay* takes a expression as an argument, and creates such a promise. The expression is not evaluated, until the function *force* is applied on it. The expression is computed only once. Its result is cached by the promise, and delivered whenever *force* is again applied on the promise.

(promise? *object*) This predicate returns *true* if its parameter is a promise.

(delay *expression*) This special form returns a promise. The evaluation of the expression is delayed until *force* is applied on the promise. The form `(delay expression)` is conceptually equivalent to `(make-promise (lambda () expression))`, where *make-promise* is defined as :

```
? (define make-promise
?     (lambda (procedure)
?         (let ((result-ready? #f)
?               (result #f))
?              (lambda ()
?                  (if result-ready?
?                      result
?                      (let ((x (procedure)))
?                           (if result-ready?
?                               result
?                               (begin (set! result-ready #t)
?                                      (set! result x)
?                                      result)))))))))
=>  make-promise
```

(force *promise*) This function returns the value associated with the promise. This value is first computed if this has not yet been done. The following session shows a promise in action.

```
? (define p (delay (+ 3 a)))
=>  p
? (promise? p)
=>  #t
(define a 5)
=>  a
? a
=>  5
? (define a 8)
=>  a
? a
=>  8
```

```
? (force p)
=>  11
? (define a 12)
=>  a
? a
=>  12
? (force p)
=>  11
```

2.1.4.16 Continuations

At each instant of the execution of a computation, the state of the computer can be represented by three objects :

the current computation which can be assimilated to a procedure without parameter giving a result ;

the current environment which describes the set of visible bindings for the *current computation*, and

the current continuation which corresponds to the operations left to do to achieve the task, and which can be assimilated to a one argument procedure waiting for the result of the *current computation*.

Misc provides access to these objects. Continuations are first class objects, assimilated to procedures.

(continuation? *object*) This predicate returns *true* if its parameter is a continuation.

```
? (call-with-current-continuation continuation?)
=>  #t
? (call-with-current-continuation procedure?)
=>  #t
```

This last example shows that a continuation is also a procedure[2].

(call-with-current-continuation *function*)
(call/cc *function*) The function *call-with-current-continuation* (for which *call/cc* is just a shorter name) packages the current continuation as a one argument procedure. The function is then called with this procedure as parameter. The result of the execution of the function becomes the result of the *call-with-current-continuation* form.

```
? (define (foo x) (+ 2 3))
=>  foo
? (+ 1 (call-with-current-continuation foo))
=>  6
```

If the parameter of the function is used as a procedure inside the function, it acts as an escape procedure. The execution of the function is abandoned, and the parameter of the procedure becomes the result of the *call-with-current-continuation* form.

[2]These two examples are hard to understand if you haven't already read at least once this whole section.

```
? (define (bar x) (+ 2 (x 3)))
=>  bar
? (+ 1 (call-with-current-continuation bar))
=>  4
```

The continuation is a first class object, with unlimited extend. It can be stored in a variable, and may be called as many times as wanted :

```
? (define cont)
=>  cont
? (define (froboz x) (set! cont x) (+ 2 3))
=>  froboz
? (+ 1 (call-with-current-continuation froboz))
=>  6
? (cont 2)
=>  3
? (+ 3 (cont 2))
=>  3
```

(dynamic-wind *before-procedure action-procedure after-procedure*) This operation accepts three parameters, which must be *thunks* (procedures with no arguments) which are successively called in order : *before-procedure action-procedure after-procedure.* The result of the *action-procedure* becomes the result of the *dynamic-wind* form.

```
? (define (nop))
=>  nop
? (define (one) 1)
=>  one
? (dynamic-wind nop one nop)
=>  1
```

The idea behind the *dynamic-wind* form is that some actions must be accompanied with auxiliary tasks. For example, working on a file content implies first opening the file, and, at the end of the work, closing the file. If, during the work on the content of the file, the continuation is captured, and then later reinvoked, it may fails because the file hasn't been reopened. Using the *dynamic-wind* form with appropriate actions as *before* and *after* procedure takes care of performing the appropriate actions. The system :

- performs first the *after-procedure* when, because of the invocation of a continuation during the *action-procedure*, it must leave the `dynamic wind` form ;
- performs first the *before-procedure* when, because of the invocation of a continuation, it must enter the *action-procedure* of a `dynamic wind` form.

The following example shows that the *before-procedure* and *after-procedure* are correctly invoked during the execution of a saved continuation.

```
? ;;; The R5RS example
? (let ((path '()) (c #f))
?       (let ((add (lambda (s) (set! path (cons s path)))))
?             (dynamic-wind
?                 (lambda () (add 'connect))
?                  (lambda () (add (call-with-current-continuation
?                                    (lambda (c0) (set! c c0) 'talk1))
?                 (lambda () (add 'disconnect)))
?           (if (< (length path) 4)
?               (c 'talk2)
?               (reverse path))))
=>  (connect talk1 disconnect connect talk2 disconnect)
```

2.1.5 Predicates

A predicate is a procedure that returns a truth value. Its name usually ends with a question mark. Some predicates (like *pair?*, *number?*, *procedure?*, etc) recognize specific data-types, and are described in the corresponding paragraphs. The following predicates are implemented.

(eq? *object object*) This operation returns *true* if its parameters are *the same object*, *false* otherwise. "The same object" means that the same physical cell is used to represent both parameters of the procedure. Two symbols with the same name are *eq*. Two booleans, both true or false, are *eq*.This is not true for other objects like numbers or strings.

```
? (eq? 'toto 'toto)
=>  #t
? (eq? 33 33)
=>  #f
? (eq? "Hi" "Hi")
=>  #f
? (eq? (symbol->string 'toto) (symbol->string 'toto))
=>  #f
? (define str "Hello")
=>  str
? (eq? str str)
=>  #t
```

(eqv? *object object*) This operation returns *true* if its parameters are *equivalent*, *false* otherwise. Objects *eq* are *equivalent*. Numbers numerically equal are equivalent. Two pairs (resulting from different applications of the *cons* procedure) are not *equivalent*.

```
? (eqv? 'toto 'toto)
=>  #t
? (eqv? 33 33)
=>  #t
? (eqv? "Hi" "Hi")
=>  #t
```

```
? (eqv? (cons 'a 'b) (cons 'a 'b))
=>  #f
```

(equal? *object object*) This operation returns *true* if its parameters are similar in structure and values. *Equivalent* objects are *equal*. Two pairs are *equal* if their *cars* are *equal* and their *cdr* are *equal*.

```
? (equal? 33 33)
=>  #t
? (equal? "Hi" "Hi")
=>  #t
? (equal? (cons 'a 'b) (cons 'a 'b))
=>  #t
```

(memq *object list*)
(memv *object list*)
(member *object list*) These predicates look for the first parameter (any object), in the list passed as the second parameter. The result is the part of the list whose first element is the object searched, or *false* if the object is not found in the list. The three predicates differ by the comparison function used to decide whether or not the object is in the list : *memq* uses *eq?*, *memv* uses *eqv?*, and *member* uses *equal*.

```
? (memq 'a '(a b c))
=>  (a b c)
? (memq 'c '(a b c))
=>  (c)
? (memq 'e'(a b c))
=>  #f
? (memq '101 '(100 101 102))
=>  #f
? (memv '101 '(100 101 102))
=>  (101 102)
? (memv '(a) '((a) (b) (c)))
=>  #f
? (member '(a) '((a) (b) (c)))
=>  ((a) (b) (c))
```

(assq *object list*)
(assv *object list*)
(assoc *object list*) These predicates look for the first parameter (any object), in the *associative list* passed as the second parameter. An associative list is a list of pairs, in which the first element is a designation (usually a symbol) of something, and the second element is an associated value. The following list is used to demonstrate the concepts and the associated predicates.

```
? al
=>  ((a . 1) (b . 2) (c . 3) (d 4) (e f) (g h) (5 i) (6 j) (k "hello")
("world" . l) ((m n) o p) ((q r s) (t u v)) (w x y) (z) (hello "Hi")
(world 333))
```

The predicates examine the successive pairs of the list, and return the first pair where the searched object appears as the first element. The three predicates differ by the comparison function used to decide whether or not the object is in the list : *assq* uses *eq?*, *assv* uses *eqv?*, and *assoc* uses *equal*.

```
? (assq 'c al)
=>  (c . 3)
? (assq 5 al)
=>  #f
? (assq 'g al)
=>  (g h)
? (assq '(q r s) al)
=>  #f
? (assq "world" al)
=>  #f
? (assv 'c al)
=>  (c . 3)
? (assv 5 al)
=>  (5 i)
? (assv 'g al)
=>  (g h)
? (assv '(q r s) al)
=>  #f
? (assv "world" al)
=>  ("world" . l)
? (assoc 5 al)
=>  (5 i)
? (assoc "world" al)
=>  ("world" . l)
? (assoc '(q r s) al)
=>  ((q r s) (t u v))
```

These predicates are typically used with cond expressions :

```
? (cond ((assoc "world" al) => cdr) (else #f))
=>  l
? (cond ((assoc '(q r s) al) => cadr) (else #f))
=>  (t u v)
```

2.1.6 MISC extensions

No "Real" SCHEME Implementation would be complete without its own set of incompatible extensions. This part of the document describes environments, threads, locks, cells and barriers.

2.1.6.1 Environments

The term of *environment* defines the set of bindings that are visible at a particular point of a program. A *binding* is the association of a name (a symbol) and a value (any object). During the execution of a program, any variable used is searched in the current environment.

An environment is actually constituted by the union of bindings (a set of symbol/value pairs), called the "current level", and a parent, which can either be empty, or be another environment. The *visible bindings* are constituted by the union of these bindings and the bindings of the parent environment.

Three environments are always available to the programmer : the *null environment*, the *report environment*, and the *interaction environment*. As a rule of thumb, the null-environment is empty, except for the binding of the *keywords* of the language. It has no parent environment. The report environment contains all the primitive operations and variables defined in the Misc language. Its parent is the null environment. Finally, the interaction environment contains the values and operations defined by the user. Its parent is the report environment.

The following operations are related to the concepts of environments.

(environment? *object*) This predicate returns *true* if its parameter is an environment.

```
? (environment? (current-environment))
=>  #t
```

(null-environment *{ object }*) This procedure returns an environment which is empty, except for the binding of all syntactic keyword defined in the language. The parent of this environment is empty. The optional parameter is not used.

```
? (null-environment)
=>  #<ENV:389:0>
? (environment-parent (null-environment))
=>  ()
```

(scheme-report-environment *{ object }*) This procedure returns an environment which contains the primitive operations of the system. Its parent is the *null environment*. The optional parameter is not used.

```
? (scheme-report-environment)
=>  #<ENV:914:69>
? (environment-parent (scheme-report-environment))
=>  #<ENV:389:0>
? (length (environment-bindings (scheme-report-environment)))
=>  159
```

(interaction-environment *{ object }*) This procedure returns an environment which contains the user's top level bindings. Its parent is the *scheme report environment*. The optional parameter is not used.

```
? (interaction-environment)
=>  #<ENV:917:70>
? (environment-parent (interaction-environment))
=>  #<ENV:914:69>
```

(define *identifier {object}*) This special form *defines* a *new binding*, the identifier being associated with the value of the second parameter either in the interaction environment when the form is not used inside a lambda expression, or in the environment of the lambda-expression when the form is used inside a lambda expression.

```
? (let ((a 3) (b 5))
?        (define c (+ a b)))
=>  c
? c
=>  8
```

When no value is specified as the second parameter, the identifier is associated with an unspecified value.

(set! *identifier object*) This special form *modifies* an *existing binding* visible in the current environment, to replace the value associated with the identifier in this environment by the value passed as a second parameter. It is an error to modify a binding which is not defined.

```
? c
=>  8
? (set! c 12)
=>  12
? c
=>  12
```

(let ((*identifier*$_1$ *expression*$_1$) (*identifier*$_2$ *expression*$_2$) ... (*identifier*$_n$ *expression*$_n$)) *sequence*) This special form establishes a new environment, with local variables, in which a *sequence* of instructions is evaluated.

More precisely, the system :

1. computes the values of *expression*$_1$ to *expression*$_n$;
2. establishes a new environment that extends the current environment by adding the *identifier*$_1$ to *identifier*$_n$, bound with the values of *expression*$_1$ to *expression*$_n$ respectively. Variables defined in this new environment may shadow external variables of the same name ;
3. evaluates in sequence the expressions of the *sequence ;*
4. discards the new environment and returns the result of the last expression of the *sequence*.

```
? (let ((a 3) (b 5) (c 0))
?      (set! c (+ a b)) (* c c))
=>  64
```

Note that since the evaluation of the values of the local variables is done outside of the environment, these expressions can reference external variables of the same name :

```
? (define a 3)
=>  a
? (define b 5)
=>  b
? (let ((a b) (b (+ a b)))
?      (* a b))
=>  40
? (* a b)
=>  15
```

The following example is another illustration of the use of the *let* form :

```
? (let ((car (lambda (x)
?                 (if (pair? x) (car x) x))))
?      (list (car '(a b c)) (car '()) (car 3)))
=>  (a () 3)
```

(letrec ((*identifier*$_1$ *expression*$_1$) (*identifier*$_2$ *expression*$_2$) ... (*identifier*$_n$ *expression*$_n$)) *sequence*)
This special form establishes a new environment, with local variables, in which a *sequence* of instructions is evaluated.

More precisely, the system :

1. establishes a new environment that extends the current environment by adding the *identifier*$_1$ to *identifier*$_n$, bound with undefined values. Variables defined in this new environment may shadow external variables of the same name ;

2. computes the values of *expression*$_1$ to *expression*$_n$;

3. sets the *identifier*$_1$ to *identifier*$_n$ to the values of *expression*$_1$ to *expression*$_n$ respectively.

4. evaluates in sequence the expressions of the *sequence ;*

5. discards the new environment and returns the result of the last expression of the *sequence*.

The *letrec* form is therefore similar to the *let* form, the difference being that the new environment is established before the evaluation of the *expression*$_i$. These expressions can't access shadowed external variables ; on the other hand, the *letrec* form makes possible to define recursive and mutually recursive procedures. These examples show the differences with the *let* form :

```
? (letrec ((fact (lambda (n)
?                        (if (= n 0) 1 (* n (fact (- n 1)))))))
?         (fact 8))
=>  40320
? (letrec ((a b) (b (+ a b))) (* a b))
Error : b undefined
```

(let* ((*identifier*$_1$ *expression*$_1$) (*identifier*$_2$ *expression*$_2$) ... (*identifier*$_n$ *expression*$_n$)) *sequence*) This special form establishes a new environment, with local variables, in which a *sequence* of instructions is evaluated. The form is equivalent to the succession of nested forms :

(let ((*identifier*$_1$ *expression*$_1$))
  (let ((*identifier*$_2$ *expression*$_2$))
    ...
      (let ((*identifier*$_n$ *expression*$_n$))
        sequence)...))

An example :

```
? (let* ((a 3)(b (+ a 2))(c (* a b)))
?       (list a b c))
=>  (3 5 15)
```

(environment-bindings *environment*) This procedure returns an associative list of symbol/value pairs representing the bindings of the current level of the environment passed as a parameter. This list doesn't include the bindings of the parent environment. Note that this list is just a representation of the environment, not the actual set of bindings. Operations on this list do not modify the bindings of the environment.

```
? (define toto 5)
=>  toto
? (define truc "Hello")
=>  truc
? (environment-bindings (interaction-environment))
=>  ((truc . "Hello") (toto . 5) (it . truc))
```

(environment-parent *environment*) This procedure returns the parent of the current environment.

```
? (eq? (scheme-report-environment)
?      (environment-parent (interaction-environment)))
=>  #t
```

(current-environment *{object}*) This procedure returns the current environment, i.e. the environment visible at this point in the program.

```
? (define ev
?     (let ((a 3) (b 5))
?           (set! a (+ a b))
?           (current-environment)))
=>  ev
? (environment-bindings ev)
=>  ((a . 8) (b . 5))
```

If a parameter is specified, the function returns the environment associated with the parameter. In the current version of MISC, the parameter can only be a defined function, i.e. a lambda expression.

(bound? *symbol {environment}*) This procedure indicates if its first parameter, a symbol, is defined or not in the environment defined by the second parameter (or in the current environment if just one parameter is specified).

```
? (bound? 'toto)
;=>  #f
? (define toto 3)
;=>  toto
? (bound? 'toto)
;=>  #t
? (bound? '+)
;=>  #t
? (bound? 'if)
;=>  #t
```

(eval *object {environment}*) This procedure evaluates its first parameter (an object representing a expression) in the environment defined by its second parameter, which must be an environment.

```
? (define ev
?     (let ((a 3) (b 5))
?           (current-environment)))
=>  ev
? (eval '(+ b 2) ev)
=>  7
```

If no environment is specified, the form is evaluated in the current environment.

(compile *object {environment}*) This procedure *compiles* its first parameter (an object representing a SCHEME expression) in the environment defined by its second parameter, which must be an environment. This procedure actually wraps its argument inside a lambda expression, such that **`(compile exp env)`** is equivalent to **`(eval (list 'lambda '() exp) env)`**. The result is a *thunk*, i.e. a function with no parameter.

```
? (define ev
?     (let ((a 3) (b 5))
?           (current-environment)))
=>  ev
? (define fun (compile '(+ b 2) ev))
=>  fun
? (fun)
=>  7
```

If no environment is specified, the form is evaluated in the current environment.

2.1.6.2 Threads

MISC provides some primitive operations that allows the user to create light-processes, designated as *threads*. A thread can be viewed as a computation in progress. When the process is over, the thread delivers a result.

A thread is characterized by :

- a name (usually, a symbol)
- an identifier (a unique number)
- a status
- a current value
- a time slice

A thread can be created (make-thread), prepared for a computation (thread-load), and made active (thread-run). Threads can be synchronized by various means : locks, cells and barriers.

(thread? *object*) This predicate returns *true* if its parameter is a thread.

```
? (thread? (current-thread))
=>  #t
```

(make-thread *symbol*) This function returns a newly created thread, whose name is defined by the symbol passed as a parameter. The thread is created in an inactive state.

(load-thread! *thread function arguments ...*) This operation prepares the thread for the execution of the function passed as the second parameter. Parameters can be passed to the function as additional parameters of the operation. The previous state of the thread is lost. The thread is left in an inactive state after this operation.

(run-thread *thread*) This operation switches to the active state the thread passed as parameter.

(current-thread) This operation returns the thread currently active.

(thread-name *thread*) This operation returns the *name* of the thread passed as parameter.

```
? (thread-name (current-thread))
=>  repl
```

(thread-id *thread*) This operation returns the *identification* of the thread passed as parameter.

```
? (thread-id (current-thread))
=>  0
```

(thread-status *thread*) This operation returns the *status* of the thread passed as parameter. The status is a symbol, that can be *init*, *run*, *wait*, *halt*, *end*, *error* or *undefined*.

```
? (thread-status (current-thread))
=>  run
```

(thread-value *thread*) This operation returns th value associated with the thread passed as parameter. This value is the result of the last execution associated with the thread.

```
? (define th (make-thread 'coco))
;=>  th
? (thread-name th)
;=>  coco
? (thread-id th)
;=>  2
? (load-thread! th + 2 3)
;=>  #<THREAD:coco:2>
? (thread-status th)
;=>  init
? (thread-value th)
;=>  ()
? (run-thread th)
;=>  #<THREAD:coco:2>
;=>  5
? (thread-value th)
;=>  5
```

(yield) This operation returns a empty list. It has the side-effect of putting the current process at the end of the queue of eligible processes, giving control to the next eligible process.

2.1.6.3 Locks

A lock is a *semaphore*. A number is associated with this semaphore. Locks are manipulated by the operations *obtain-lock* and *release-lock*. Each time *obtain-lock* is called, the number associated with the lock is decremented. When a thread calls *obtain-lock*, and the number associated with the lock becomes negative after the call, the thread is put in wait state, and entered in a wait queue associated with the lock. When a thread calls *release-lock*, the number associated with the lock is incremented, and if the queue associated with the lock is not empty, the first thread in the wait queue is removed from the queue and made active.

(lock? *object*) This predicate returns *true* if its parameter is a lock.

(make-lock *number*) This operation creates a new lock, initialized with the value of its parameter which must be a non negative integer.

(obtain-lock *lock*) This operation decrements the number associated with the lock. If the new value of the number is negative, the thread is halted until a sufficient number of *release-lock* operations are issued on the lock.

(release-lock *lock*) This operation increments the number associated with the lock. If the threads wait queue associated with the lock is non empty, the first thread in the wait queue is removed from the queue and made active.

(lock-value *lock*) This operation returns the integer value associated with a lock.

2.1.6.4 Cells

A cell is a place in memory than can hold a value. Threads can be made to wait until the value becomes available.

(cell? *object*) This predicate returns *true* if its parameter is a cell.

(make-cell) This operation creates a new cell containing no value.

(cell-value *cell*) This operation returns the value associated with the cell. If no value is associated with the cell, the thread is halted until some other thread set the value of the cell.

(cell-set! *cell object*) This operation sets the value associated with the cell. If others threads were waiting for the value of the cell, these threads are all removed from the wait queue associated with the cell and made active.

(cell-value-available? *cell*) This predicate returns *true* if its parameter is a cell with a value available.

2.1.6.5 Barriers

A barrier is yet another way to synchronize threads. As a lock, a barrier has a counter associated with. When a thread calls wait-barrier, this counter is decremented. If the value of the counter is positive, the thread is put in wait state. If the value of the counter reaches 0, all the threads waiting at the barrier are made active, and the initial value of the counter is restored. A barrier is therefore a means to synchronize processes, none of these processes being masters or slaves.

(barrier? *object*) This predicate returns *true* if its parameter is a barrier.

(make-barrier *number*) This operation creates a new barrier, the parameter being the initial value of the counter associated with the barrier.

(barrier-wait *barrier*) This operation decrements the value of the counter associated with the barrier. If this value becomes equal to 0, the current thread and all other threads waiting at the barrier are made active. If the number is positive, the current thread is put in wait state.

2.1.6.6 Error trapping

(error *string*) This operation interrupts the current computation and prints an error message containing the string passed as a parameter to the function :

```
? (error "A message")
Error : A message
? (list 1 2 (error "an error") 3 4)
Error : an error
```

Note that the result is a *Java error*. See **try** on this page and **error?** on the next page.

(try *thunk0 thunk1*) This operation executes *thunk0*, a procedure without parameter, and returns its result as the result of the try operation. If an error arises during the execution of *thunk0*, the error is packaged as an item, and the one parameter procedure *thunk1* is executed with the packaged error passed as parameter. The result of *thunk1* is then returned as the result of the *try* form.

In this first example, no error arises, and the procedure *thunk1* is not called :

```
? (try (lambda () (display "Hello\n"))
?      (lambda (e) (display (string "Hum" e))))
Hello
=> "Hello\n"
```

In this second example, an error arises during the execution of *thunk0*. The procedure *thunk1* is therefore invoked, with the error passed as an argument.

```
? (define err)
=> err
? (try (lambda () (display "Hello\n")
?                 (+ 2 "abc") (display "Hum\n"))
?      (lambda (e) (set! err e) (display "Error found\n")))
Hello
Error found
=> "Error found\n"
? err
=> "Sch.SchRunTimeException: integer expected"
? (error? err)
=>  #t
```

2.1.6.7 Reflection

The concept of «reflection» refers to the possibility to access, manipulate and use the objects that constitute or implement the language itself. Misc provides access to all objects and concepts of scheme (such as functions, environments or continuations), but also to the implementation language, since Java itself provides reflection methods. The following operations are available to access the java objects :

(java.get-class *string*) This operation returns the *class* whose name is passed as parameter, in the form of a string or a symbol.

```
? (define Integer (java.get-class "java.lang.Integer"))
=> Integer
? Integer
=> "class java.lang.Integer"
```

(class? *object*) This operation returns *true* if its parameter is a Java class.

```
? (class? Integer)
=> #t
```

(object? *object*) This operation returns *true* if its parameter is a Java *object*.

```
? (object? Integer)
=> #t
? (object? "abcd")
=> #t
? (object? 333)
=> #f
```

(error? *object*) This operation returns *true* if its parameter is an instance of a Java error. Such an object can be returned when an error arises while attempting to apply a method, a constructor or a field. It can also be the result of trapping a Misc run-time error with the **try** procedure (c.f. **try** on the preceding page).

(java.classof *object*) The parameter may be any object. The procedure returns the java class of the object :

```
? (java.classof "abc")
=> "class java.lang.String"
? (java.classof Integer)
=> "class java.lang.Class"
? (type (java.classof Integer))
=>  java:java.lang.Class
```

The operation is not fully consistent with the notion of reflection, since it returns the class **Sch.Sch** for some scheme values such as integers or booleans, while we could expect to get classes **java.lang.Integer** or **java.lang.Boolean** instead :

```
? (java.classof 23)
=> "class Sch.Sch"
? (java.classof #t)
=> "class Sch.Sch"
```

(java.name *object*) This operation returns the name of the object as a character string.

```
? (java.name Integer)
;=>  "java.lang.Integer"
? (type (java.name Integer))
;=>  string
? (java.name 23)
;=>  ""
```

(java.constructors *object {string}*) The first parameter must be a Java class. This operation returns the list of all the *constructors* of the class if there is just one parameter, or the list of the constructors of the class that are filtered by the second parameter, which must be a regular expression :

```
? (java.constructors Integer)
=> ("public java.lang.Integer(java.lang.String) throws
java.lang.NumberFormatException" "public java.lang.Integer(int)")
? (java.constructors Integer "*(int)*")
=> ("public java.lang.Integer(int)")
```

(java.methods *object {string}*) The first parameter must be a Java class. This operation returns the list of all the *methods* of the class if there is just one parameter, or the list of the methods of the class that are filtered by the second parameter, which must be a regular expression :

```
? (java.methods Integer "*HexString*")
=> ("public static java.lang.String
java.lang.Integer.toHexString(int)")
```

(java.fields *object {string}*) The first parameter must be a Java class. This operation returns the list of all the *fields* of the class if there is just one parameter, or the list of the fields of the class that are filtered by the second parameter, which must be a regular expression :

```
? (java.fields Integer)
=> ("public static final java.lang.Class java.lang.Integer.TYPE"
"public static final int java.lang.Integer.MAX_VALUE"
"public static final int java.lang.Integer.MIN_VALUE")
```

2.1.6.8 Using reflection in java

Objects created by the procedures *java.constructors*, *java.methods* and *java.fields* are considered as procedures by the scheme interpreter. They can be directly invoked from scheme as procedures with parameters. We will therefore use the terms **method procedure**, **constructor procedure** and **field procedure** to reference such procedures.

Class methods (or static methods) These methods can be directly applied on a set of parameters, as expected by the method :

```
? (define Integer (java.get-class "java.lang.Integer"))
=> Integer
? (define toOct (car (java.methods Integer "*toOct*")))
=> toOct
? (toOct 1023)
=> "1777"
```

Here, we first access the class **java.lang.Integer**, which will be referenced as the variable **Integer**. Then we access the static method toOctalString. Note that java.methods always return a list of methods. Since we are sure that just one method has been selected, we just take, with car, the first element of the list. Finally, the method is used as a procedure, and applied to a number. In this case, the result is a string representing the octal value of the number.

The operation java.invoke can also be used to invoke a method on an object with parameters.

(java.invoke *method object par1 par2 ... parn*) This operation invokes the specified *method* on *object*, with parameters *par1*, *par2*, etc.

```
? (define Math (java.get-class 'java.lang.Math))
;=>  Math
? (define rand (car (java.methods Math "*random(*")))
;=>  rand
? (define res (java.invoke rand ()))
;=>  res
? res
;=>  "0.93461781473136"
? (object? res)
;=>  #t
? (type res)
;=>  java:java.lang.Double
? (string res)
;=>  "0.93461781473136"
? (define log (car (java.methods Math "*.log(*")))
;=>  log
? (java.invoke log () res)
;=>  "-0.06761758755427073"
```

Instance methods Like class methods, instances methods can be extracted from a class with the procedure **java.methods**. Such methods must be applied to an object of the corresponding class, which is passed as the first parameter of the scheme procedure. Other parameters are the actual parameters of the Java method.

Constructors Constructor procedures are obtained with the procedure **java.constructors**. They can be applied to 0 or more parameters, according to their definition, to obtain an object of the desired class.

```
? (define java.lang.Double (java.get-class 'java.lang.Double))
=> java.lang.Double
```

```
? (define StD (car (java.constructors java.lang.Double "*(*String)*")))
=> StD
? (StD "0.000023E+12")
=> "2.3E7"
```

In this example, we access the class **java.lang.Double**, get the constructor taking a **String** as parameter, and use this constructor to create an instance of **Double**. Since the type "double" is unknown in Misc, the system uses "asString" to get a printable value of the object, showing in this case that the object has been correctly built.

Class fields (or static fields) Fields are also viewed as scheme procedures, that can be used either to access or to modify the value of a field.

```
? (define MxV (car (java.fields java.lang.Double "*MAX_VALUE*")))
=> MxV
? (MxV)
=> "1.7976931348623157E308"
```

We select here the static field named MAX_VALUE of the class **Double**, and access its value by a call to the field procedure without parameter. Changing the value of the field can be done by using the field procedure with one parameter, this value being used to replace the previous value of the field. The result is the previous value of the field. As an example :

```
? (MxV 0)
=> "java.lang.IllegalAccessException: field is final"
? (error? (MxV 0))
=> #t
```

is the correct syntax to change the value of the field MAX_VALUE, except that this field is declared *final* and can't be modified.

In the next example, we access the static field named `trace` in the class **Sch** (actually, inherited from **Globaldata**), and obtain a procedure, **tra**, which is equivalent to the **trace** procedure of the system.

```
? (define Sch.Sch (java.get-class 'Sch.Sch))
=> Sch.Sch
? (define tra (car (java.fields Sch.Sch "*trace*")))
=> tra
? (tra)
=> 0
? (tra 1)
=> 0
? (tra)
=> 1
```

Instance Fields In the same spirit, an instance field of an object can be accessed by using the corresponding *field procedure*, and providing it an object of the expected class as the first parameter. Modifying the field is done by providing as the second parameter the new value of the field (the procedure returns the previous value of the field).

Notes All these procedures may return errors, resulting from the usage of incorrect arguments or the lack of permissions (for example, private methods can't be invoked).

2.1.6.9 Dynamic loading

Misc allows the dynamic loading of compiled modules. In the following example, the `DemoOps.java` is such a module. It implements a single operation, `demo`, that takes no argument and always returns «true».

(load-misc-module *module*) This operation dynamically loads the specified module, which must be accessible by the java machine. The interpeter, after loading, invokes the «dcl» class method, which allows the module to declare its entry points.

```
? (demo)
Error : demo undefined
? (load-misc-module "DemoOps")
;=>  "class DemoOps"
?  (demo)
;=>  #t
? (demo 2 3 4)
Error : no argument expected
? (list 1 2 (demo) 3 4 5 6 7 8)
;=>  (1 2 #t 3 4 5 6 7 8)
```

2.1.6.10 Some other functions

Here is a set of yet unqualified functions of the system.

(type *object*) This operation returns a symbol characteristic of the type of the object passed as parameter :

```
? (type 3)
=>  number
? (type "abcd")
=>  string
? (type car)
=>  subr
? (type '(a b c))
=>  pair
? (type (read (open-input-sink)))
=>  end-of-file
```

(quit)
(stop) These two equivalent operations are escape procedures that abort the current computation and return to the read-eval-print loop.

```
? (list 1 2 (quit) 3 4)
?
```

(end)
(exit) These two equivalent procedures are escape procedures that abort the current computation and exit from the MISC interpreter.

```
? (list 1 2 (exit) 3 4)
; Cells use : 1131/8192, 0 gc.
Misc ended
```

(idt *val*) This procedure returns its parameter unchanged, and is equivalent to `(lambda (x) x)` ; it may be useful in some specific cases.

2.1.6.11 Debugging

This section describes some debugging tools, that should not be used for casual programming in Scheme.

(gc) This operation activates the garbage collector, which collects unused cells, and returns the number of cells available. New cells are dynamically added when the numbers of free cells is less than a percentage of the total number of cells after a garbage collection.

```
? (gc)
=>  7127
```

(debug {int}) This operation, with no parameter, returns the value of the debug flag, and with an integer parameter sets this value. The value of the debug flag is an or-combination of the following values :

1 This value puts the memory management in «starving mode» : after the allocation of a memory cell, the empty cell list is cleared, so that the next allocation request will trigger the garbage collector ; this is specially useful for debugging purpose, but will drastically slow the interpreter.

2 This value will trigger a garbage collector before the execution of each primitive of the interpreter ; this is useful to catch unprotected temporary values, and will also slow down the interpreter.

4 This value will trace some operations of the memory management.

(dump *object*) This operation prints the contents of the object passed as a parameter, and returns the object itself. It is essentially similar to the write operation, but performs a skip to the next line after printing.

```
? (dump "hello")
"hello"
;=> "hello"
? (dump '(a b c))
(a b c)
;=>  (a b c)
? (dump (cell 1))
#<UNDEF>
;=>  #<UNDEF>
```

(cell-address *object*) This operation returns the cell number of the object passed as parameter. There is no garantee that some value is always represented by the same cell, except for a few predefined objects, such as null, true, false, etc.

```
? (cell-address "hello")
;=> 2219
? (cell-address "world")
;=> 2237
? (cell-address #t)
;=>  3
? (cell-address 'cons)
;=>  404
```

(cell *number*) This operation returns the object located at the specified cell number. The parameter should be a non negative integer, not greater than the number of memory cells used.

```
? (cell 2219)
;=>  "hello"
? (cell 3)
;=>  #t
?  (cell 2)
;=>  #f
? (cell 429)
;=>  list
```

(cell-dump *begin {end}*) This operation prints the contents of the cells starting at the location specified by begin, up to the location defined by end. The default value for end is the begin value. Aside from the side effect of printing this content, the result of the operation is the empty list.

```
? (cell-dump 2)
2:c:0:0:()
;=>  ()
? (cell-dump 0 4)
0:0:0:0:()
1:12:0:0:()
2:c:0:0:()
3:c:0:0:()
4:9:4:0:()
;=>  ()
? (cell-address '(a b c))
;=>  2584
? (cell-dump 2584)
a18:2:6fd:a1b:()
;=>  ()
? (cell-address "hello")
;=>  2631
? (cell-dump 2631)
```

```
a47:5:0:0:hello
;=>  ()
? (define Math (java.get-class 'java.lang.Math))
;=>  Math
? (cell-address Math)
;=>  2721
? (cell-dump 2721)
aa1:1b:0:0:class java.lang.Math
;=>  ()
```

(trace *{number}*) This operation reads (with no parameters) or sets (with one parameter) the trace flags of the interpreter. Useful values for the trace are :

1 this prints information about execution time (the "Ticks") and space used ("Stack" entries and number of "Cells") by the evaluation of expressions entered at the terminal.

```
? (+ 2 3)
=>  5
; Stack use : 5/16, 3/16
; Cells use : 1/8192, 0 gc.
; Ticks use : 7
```

This addition has used 5 entries in the *value stack*, 3 in the *control stack*, and 1 cell out of a total of 8192. It was executed in 7 elementary operations of the Misc machine.

2 this prints the contents of the control and value stacks in case of error.

```
? (list 1 2 (+ 3 "abc") 4 5)
 Value Stack (2)
     0   5
     1   4
 Control Stack (6)
     0   [END]
     1   [F5]
     2   {list}
     3   1
     4   2
     5   [F2]
Error : integer expected
```

4 this prints the internal representation of the instruction before its execution.

```
? (+ 2 3)
-> ([F2] {+} 2 3)
=>  5
```

8 this traces each operation of the Misc machine.

```
? (+ 2 3)
   [ENV] ----> 71 {{1028:70}}
   ([F2] {+} 2 3)
   3
   2
   {+}
   [F2]
   [END]
=>  5
```

These values can be added to obtain the combination of the different functions.

2.1.6.12 Some unclassified objects

(the-undefined-object) This operation returns the undefined-object, wich is the default value of any undefined variable :

```
? titi
Error : titi undefined
? (define titi)
=>  titi
? titi
=>  ()
? (set! titi 3)
=>  3
? titi
=>  3
? (set! titi (the-undefined-object))
=>  #<UNDEF>
? titi
Error : titi undefined
```

it This variable contains the value of the last result computed in interactive mode. It may be useful for a result that took a long time to be computed, but was not assigned to a variable.

```
? (+ 2 3 5)
=> 10
? it
=> 10
? (* it 3)
=> 30
? it
=> 30
? (set! x it)
=> 30
```

Chapter 3

Examples of MISC Programs

This sections gives classical and typical examples of the use of Scheme as a general purpose programming language.

3.1 Sorting

This is a classical *merge sort* algorithm. Sorting a list of numbers is done by `(sort list <)`.

```
(define (sort l less)
    (define (merge l1 l2)
        (if (null? l1)
            l2
            (if (null? l2)
                l1
                (if (less (car l1) (car l2))
                    (cons (car l1) (merge (cdr l1) l2))
                    (cons (car l2) (merge l1 (cdr l2)))))))
    (define (sort-aux l)
        (if (or (null? l) (null? (cdr l)))
            l
            (cut l '() '())))
    (define (cut l l1 l2)
        (if (null? l)
            (merge (sort-aux l1) (sort-aux l2))
            (cut (cdr l) (cons (car l) l2) l1)))
    (sort-aux l))
```

This is an example of use.

```
? (define l '(2 9 4 6 3 9 1 20 5 2 8 3 5 10 0 4 9 7 3 2 11))
=>  l
? (sort l <)
=>  (0 1 2 2 2 3 3 3 4 4 5 5 6 7 8 9 9 9 10 11 20)
```

3.2 Exploring a tree

It is sometimes necessary to explore an arbitrary data structure, looking for elements satisfying some predicate. The following function, *make-walker*, given a tree and a predicate, generates a function that explores the tree looking for elements satisfying the given predicate. Moreover, the function generates a single result each time it is called, giving the value *false* at the end of the exploration.

```
(define (make-walker T pred?)
    (define (explore S cont!)
         (if (pair? S)
             (explore (car S) (lambda () (explore (cdr S) cont!)))
             (if (null? S)
                 (cont!)
                 (if (pred? S)
                     (begin (set! glob-cont! cont!) S)
                     (cont!)))))
    (define (glob-cont!)
         (explore T (lambda () #f)))
    (lambda () (glob-cont!)))
```

The following data structure is used for the tests :

```
(define Tree '(1 2 3 -9 3 (5 6 -2 8 -7 3)
        (9 (11 -3 12) 7 11 (13 (15 (17 ((19 20) -3 -7)
        -11)))) 6 56 -17 (((((((3))))) 7 -8)))))
```

We generate now three walkers : `toto` looks for *negative* elements, `titi` for *odd* elements, and `tutu` for elements congruent to 1 modulo 5.

```
? (define toto (make-walker Tree negative?))
=>  toto
? (define titi (make-walker Tree odd?))
=>  titi
? (define tutu (make-walker Tree (lambda (x) (= (modulo x 5) 1))))
=>  tutu
```

Here are the first calls of these three functions :

```
? (toto)
=>  -9
? (titi)
=>  1
? (tutu)
=>  1
? (list (toto) (titi) (tutu))
=>  (-2 3 -9)
? (list (toto) (titi) (tutu))
=>  (-7 -9 6)
? (list (toto) (titi) (tutu))
=>  (-3 3 11)
```

3.3 An «expert system»

This is a micro expert system, based on the article «Introduction aux systèmes experts» [3], which describes a botanic data-base, where the notation **fleur** means «the plant has flowers», where $\bigwedge$ is the logical *and*, and $\neg$ is the negation. This is a set of propositions :

a) fleur $\bigwedge$ graine $\Longrightarrow$ phanérogame
b) phanérogame $\bigwedge$ graine-nue $\Longrightarrow$ sapin
c) phanérogame $\bigwedge$ 1-cotylédone $\Longrightarrow$ monocotylédone
d) phanérogame $\bigwedge$ 2-cotylédone $\Longrightarrow$ dicotylédone
e) monocotylédone $\bigwedge$ rhizome $\Longrightarrow$ muguet
f) dicotylédone $\Longrightarrow$ anémone
g) monocotylédone $\bigwedge \neg$ rhizome $\Longrightarrow$ lilas
h) feuille $\bigwedge \neg$ fleur $\Longrightarrow$ cryptogame
i) cryptogame $\bigwedge \neg$ racine $\Longrightarrow$ mousse
j) cryptogame $\bigwedge$ racine $\Longrightarrow$ fougère
k) $\neg$ feuille $\bigwedge$ plante $\Longrightarrow$ thallophyte
l) thallophyte $\bigwedge$ chlorophylle $\Longrightarrow$ algue
m) thallophyte $\bigwedge \neg$ chlorophylle $\Longrightarrow$ champignon
n) $\neg$ feuille $\bigwedge \neg$ fleur $\bigwedge \neg$ plante $\Longrightarrow$ collibacille

Some of these results are intermediate stages of the determination process, others are conclusions :

sapin	muguet	anémone	lilas	mousse
fougère	algue	champignon	collibacille	

Here is a definition of the data-base, where the symbol − is used to denote negation :

```
(define BDD '(
     phanérogame (fleur graine)
     sapin (phanérogame graine-nue)
     monocotyledone (phanérogame 1-cotyledone)
     dicotyledone (phanérogame 2-cotyledone)
     muguet (monocotyledone rhizome)
     anemone (dicotyledone)
     lilas (monocotyledone - rhizome)
     cryptogame (feuille - fleur)
     mousse (cryptogame - racine)
     fougere (cryptogame racine)
     thallophyte (- feuille plante)
     algue (thallophyte chlorophylle)
     champignon (thallophyte - chlorophylle)
     collibacille (- feuille - fleur - plante)))
```

This is the set of goals, the possible deductions :

```
(define BUTS '(
       sapin muguet anemone lilas mousse
       fougere algue champignon collibacille))
```

Let's suppose our specimen has the following characteristics :

```
rhizome fleur graine 1-cotyledone
```

The program tries every hypothesis, using known facts, adding to this set of knowledge every deduction it can make. According to our data-base, this is the set of possible conclusions :

```
(fleur graine) ⟹ phanérogame
(phanérogame 1-cotyledone) ⟹ monocotyledone
(monocotyledone rhizome) ⟹ muguet
```

Muguet is one of the goals ; the nature of the specimen is therefore determined.

Here is the SCHEME program :

```
(define (deduire vrais faux BDD BUTS)
    (define (tente truc vrais faux)
        (define (valide? proposition vrais faux)
            (or (null? proposition)
                (and (eq? '- (car proposition))
                     (memq (cadr proposition) faux)
                     (valide? (cddr proposition) vrais faux))
                (and (memq (car proposition) vrais)
                     (valide? (cdr proposition) vrais faux))))
        (if (valide? (cadr truc) vrais faux)
            (car truc)
            #f))
    (define (balaye base vrais faux)
        (if (null? base)
            #f
            (essai (tente base vrais faux) base vrais faux)))
    (define (essai fait base vrais faux)
        (if (and fait (not (memq fait vrais)))
            fait
            (balaye (cddr base) vrais faux)))
    (define (itere vrais faux)
        (itere2 (balaye BDD vrais faux) vrais faux))
    (define (itere2 fait vrais faux)
        (if (not fait)
            #f
            (if (memq fait BUTS)
                fait
                (itere (cons fait vrais) faux))))
    (itere vrais faux))
```

Finally, here are some examples of the use of the program :

```
? (deduire '(rhizome fleur graine 1-cotyledone) '() BDD BUTS)
```

```
=>  muguet
? (deduire '(fleur graine 2-cotyledone) '() BDD BUTS)
=>  anemone
? (deduire '(thallophyte) '(chlorophylle) BDD BUTS)
=>  champignon
? (deduire '(chlorophylle) '(fleur feuille rhizome) BDD BUTS)
=>  #f
```

The last example is a case where no conclusion can be drawn from the proposed premises.

Bibliography

[1] Harold Abelson, Gerald Jay Sussman, and Julie Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, Cambridge, Mass., 1985.

[2] Harold Abelson, Gerald Jay Sussman, and Julie Sussman. *Structure et Interprétation des Programmes Informatiques*. InterEditions, Paris, France, 1989.

[3] M. Gondran. Introduction aux systèmes experts. *Bulletin de la Direction des Etudes et Recherches d'E.D.F., Série C, Mathématiques Informatiques*, (2), 1983.

[4] Jean-Jacques Girardot. Misc Internals Manual. Technical report, École des Mines, Saint-Étienne, France, 2001.

[5] Jean-Jacques Girardot. Misc User's Manual. Technical report, École des Mines, Saint-Étienne, France, 2001.

[6] R5RS. Revised ^{5} Report on the Algorithmic Language Scheme. Technical report, 1998.

[7] Christian Rolland. *LaTeX2ε, guide pratique*. Addison-Wesley, France, Juin 1995.

[8] LyX Team. Implémentation de LyX. `http://www.lyx.org/`.

Index

(), 17
*, 12, 16
+, 11
-, 12
<, 12
<=, 12
=, 12
=>, 7, 8
>, 12
>=, 12
!=, 12
", 6
#B, 14
#D, 14
#O, 14
#X, 14
#b, 14
#d, 14
#f, 11
#o, 14
#t, 11
#x, 14
#\, 14
’, 6
‘, 6
#, 6
,, 6
;, 6
@, 6

abs, 13
Alphanumeric characters, 5
and, 7, 8
append, 20
append!, 20
apply, 30
assoc, 35
associative list, 35
assq, 35
assv, 35
at-sign, 6

backquote, 6
barrier-wait, 44
barrier?, 44
Barriers, 10, 44
begin, 7, 8
binding, 36
boolean?, 11
Booleans, 10, 11
bound?, 41

caaaar, 20
caaadr, 20
caaar, 19
caadar, 20
caaddr, 20
caadr, 19
caar, 19
cadaar, 20
cadadr, 20
cadar, 19
caddar, 20
cadddr, 20
caddr, 19
cadr, 19
call-with-current-continuation, 32
call/cc, 32
car, 10, 19
case, 7, 9
cdaaar, 20
cdaadr, 20
cdaar, 19
cdadar, 20
cdaddr, 20
cdadr, 19
cdar, 19
cddaar, 20
cddadr, 20
cddar, 19

cdddar, 20
cddddr, 20
cdddr, 19
cddr, 19
cdr, 10, 19
cell, 44, 52
cell-address, 52
cell-dump, 52
cell-set!, 44
cell-value, 44
cell-value-available?, 44
cell?, 44
Cells, 10, 44
char->integer, 14
char-ready?, 26
char?, 14
Characters, 10, 14
class, 46
class? (procedure), 46
close-input-port, 25
close-output-port, 25
comma, 6
Comments, 6
comments, 6
compile, 41
cond, 7, 8
cons, 18, 34
constructor, 47
continuation?, 32
Continuations, 10, 32
continuations, 4
copy, 22
current-environment, 40
current-input-port, 24
current-output-port, 24
current-thread, 42

debug, 51
deep-copy, 22
define, 7, 38
delay, 7, 31
demo, 50
difference, 12
display, 28
do, 7
dot, 5
double quote, 6, 14
dump, 51
dynamic-wind, 33

else, 7
Empty list, 10, 17
empty list, 20
end, 51
end-of-file object, 25
environment, 36
environment-bindings, 40
environment-parent, 40
environment?, 37
Environments, 10, 36
environments, 4
eof object, 27
eof-object?, 27
eq?, 34–36
equal, 35, 36
equal?, 35
eqv?, 9, 34–36
error, 45
error? (procedure), 46
eval, 41
even?, 13
exit, 51

false, 11
field, 47
first class citizen, 10, 28
for-each, 30
force, 31
Forms, 6
Function application, 7
future, 10

garbage collector, 4, 51
gc (procedure), 51
get-port-string, 25

idt, 51
if, 7
Input operations, 25
input-port?, 24
integer->char, 14
integer?, 11
Integers, 11
interaction-environment, 37
it, 54

java.classof (procedure), 46
java.constructors (procedure), 47
java.fields (procedure), 47
java.get-class (procedure), 46
java.invoke, 48
java.methods (procedure), 47
java.name, 47

keyword, 6
keywords, 37

lambda, 7, 28
length, 21
let, 7, 38
let*, 7, 40
letrec, 7, 39
light-weight processes, 4
list, 20
list-ref, 22
list-tail, 22
list?, 20
Lists, 20
load, 27
load-misc-module, 50
load-thread!, 42
lock, 43
lock-value, 44
lock?, 43
Locks, 10, 43

make-cell, 44
make-lock, 43
make-string, 15
make-thread, 42
make-vector, 23
map, 30
mark and sweep, 4
max, 13
maximum, 13
member, 35
memq, 35
memv, 35
method, 47
min, 13
minimum, 13
Misc, 3
Misc home page, 3
modulo, 13

negative?, 12
newline, 28
not, 11
null-environment, 37
null?, 18
number->string, 16
number?, 11
Numbers, 10, 11

object, 46
object? (procedure), 46
obtain-lock, 43
odd?, 13
open-input-file, 25
open-input-sink, 25
open-input-string, 24
open-output-file, 25
open-output-sink, 25
open-output-string, 25
open-URL, 25
opposite, 12
or, 7, 8
Output operations, 28
output-port?, 24

pair?, 18
Pairs, 10, 18
parentheses, 6
parity, 13
pattern matching, 16
peak-char, 26
port?, 24
Ports, 10, 24
positive?, 12
Predicates, 34
procedure?, 29
Procedures, 10, 28
product, 12
promise?, 31
Promises, 31
proper list, 20

quasiquote, 7
quit, 50
quotation, 7
quote, 6, 7, 10
quoted, 10
quotient, 13

read, 25
read-char, 26
Read-Eval-Print, 26
read-line, 26
read-token, 26
read-XML, 27
read-XML-token, 27
reified, 10
release-lock, 44
remainder, 13
reverse, 21
reverse!, 21
run-thread, 42

Scheme, 3
scheme-report-environment, 37
semi-colon, 6
semicolon, 6
set-car!, 19
set-cdr!, 19
set!, 7, 38
sharp sign, 6
sink-input-port, 24
sink-output-port, 24
sort, 55
Sorting, 55
Spaces, 5
Special characters, 6
Special forms, 6
stop, 50
string, 15
string->symbol, 17
string-append, 15
string-length, 15
string-match, 16
string-ref, 16
string-set!, 16
string?, 15
Strings, 10, 14
substring, 16
sum, 11
symbol, 5
symbol->string, 17
symbol?, 17
Symbols, 10, 16
Syntax, 5

tail recursive, 4
terminal-input-port, 24
terminal-output-port, 24
the-non-displayable-objet, 28
the-undefined-object, 54
thread, 42
thread-id, 42
thread-name, 42
thread-status, 43
thread-value, 43
thread?, 42
Threads, 10, 42
thunk, 41
trace, 53
trace-port, 24
true, 11
try, 45
type, 50

unquote, 7
unquote-splicing, 7

vector, 23
vector-fill, 24
vector-length, 23
vector-ref, 23
vector-set, 23
vector?, 23
Vectors, 23
vectors, 10
visible bindings, 37

write, 28

XML, 27

yield, 43

zero?, 12

Contents

1 Introduction **3**
1.1 What is MISC ? . . . 3
1.1.1 A few words . . . 3
1.1.2 About this manual . . . 3
1.2 A short introduction to MISC . . . 3
1.2.1 Getting the system . . . 3
1.2.2 Installing the system . . . 3
1.2.3 Running the system . . . 4
1.2.4 About MISC . . . 4

2 The language **5**
2.1 A short description of MISC . . . 5
2.1.1 The standard part of MISC . . . 5
2.1.1.1 Syntax . . . 5
2.1.2 Special forms . . . 7
2.1.3 Data types . . . 10
2.1.4 Operations on data types . . . 11
2.1.4.1 Booleans . . . 11
2.1.4.2 Numbers . . . 11
2.1.4.3 Integers . . . 11
2.1.4.4 Characters . . . 14
2.1.4.5 Strings . . . 14
2.1.4.6 Symbols . . . 16
2.1.4.7 Empty list . . . 17
2.1.4.8 Pairs . . . 18
2.1.4.9 Lists . . . 20
2.1.4.10 Vectors . . . 23
2.1.4.11 Ports . . . 24
2.1.4.12 Input operations . . . 25
2.1.4.13 Output operations . . . 28
2.1.4.14 Procedures . . . 28
2.1.4.15 Promises . . . 31
2.1.4.16 Continuations . . . 32
2.1.5 Predicates . . . 34
2.1.6 MISC extensions . . . 36
2.1.6.1 Environments . . . 36
2.1.6.2 Threads . . . 42

2.1.6.3 Locks . . . 43
2.1.6.4 Cells . . . 44
2.1.6.5 Barriers . . . 44
2.1.6.6 Error trapping . . . 45
2.1.6.7 Reflection . . . 45
2.1.6.8 Using reflection in java . . . 47
2.1.6.9 Dynamic loading . . . 50
2.1.6.10 Some other functions . . . 50
2.1.6.11 Debugging . . . 51
2.1.6.12 Some unclassified objects . . . 54

3 Examples of MISC Programs **55**
3.1 Sorting . . . 55
3.2 Exploring a tree . . . 56
3.3 An «expert system» . . . 57

References **61**

Index **63**

Table of contents **67**